Performance of Object-Based Semantic Real-Time Concurrency Control

Lisa Cingiser DiPippo and Victor Fay Wolfe
Department of Computer Science
University of Rhode Island
Kingston, RI 02881
lastname@cs.uri.edu

Abstract

In this paper we present the results of performance tests that were executed to demonstrate the effectiveness of an object-based semantic real-time concurrency control technique. The paper reviews the technique and the model upon which it was based. It then presents the results of several tests comparing two implementations of the semantic technique with three more traditional object-based concurrency control techniques. The results indicate that the semantic techniques maintained both transaction temporal consistency and data temporal consistency better than the traditional techniques.

1 Introduction

A real-time database is a database in which both the data and the transactions may be time-constrained. The concurrency control for such a database must therefore maintain both the *logical consistency* constraints of traditional databases, and the *temporal consistency* constraints placed on data and on transactions. Data temporal consistency requirements constrain how old a data item may be and still be considered valid. Transaction temporal consistency requirements constrain when the transaction may execute. Unfortunately, an inherent conflict exists between maintaining logical consistency and maintaining temporal consistency. For example, if a transaction T_1 is currently reading an object X, the logical consistency requirements of T_1 would not allow another transaction T_2 to write to X concurrently. However, suppose that the data stored in X will become old if T_2 does not perform its write. In this case, the logical consistency of T_1 may be sacrificed in order to maintain the temporal consistency of the object X. This violation of logical consistency may lead to some amount of imprecision in the view that T_1 has of X.

We have developed an object-based semantic concurrency control technique that can express the trade-off between temporal and logical consistency by using object semantics to define conflict between locks [DW93]. For instance, object semantics may determine that in order to maintain certain temporal consistency constraints, the logical consistency of the data or transactions must be sacrificed. More traditional object-based concurrency control techniques define conflict with less flexibility. For instance, exclusive locking defines conflict as any two concurrent accesses to the object. Techniques like read/write locking, and commutative locking [BR88] use more object semantics to define conflict. In general, the more flexibility in the definition

of conflict, the more concurrency that will be allowed by an object-locking technique. Furthermore, this increased concurrency will allow more temporal consistency constraints to be maintained.

In this paper we describe performance tests that we executed to compare our semantic concurrency control technique with other techniques that provide varying degrees of flexibility in conflict definition. The tests compare the temporal consistency provided by more traditional object locking techniques with our semantic locking technique. We measure both transaction temporal consistency and data temporal consistency to show how increased allowable concurrency affects the performance of a real-time database.

The remainder of this paper is organized as follows. Section 2 briefly describes our model for a real-time object-oriented database, and our semantic locking technique based on that model. It also describes a prototype implementation of two versions of the technique that represent specific chosen object semantics. Section 3 describes the environment in which we performed the tests comparing the object locking techniques. Section 4 describes the specific tests that we performed and presents the results of the tests. Finally, Section 5 concludes with remarks summarizing the test results and describes possible future work.

2 Object-based Semantic Concurrency Control

In this section we briefly describe our model of a real-time object-oriented database, called RTSORAC (**R**eal-**T**ime **S**emantic **O**bjects, **R**elationships **A**nd **C**onstraints) [PDPW94]. It continues with an overview of our semantic locking technique for real-time concurrency control. This review is important for the understanding of the tests that we performed. A full description of both the model and the technique can be found in [DW93, DW].

2.1 RTSORAC Model

The RTSORAC model incorporates time into objects and transactions to allow for explicit specification of data temporal consistency as well as transaction temporal consistency. The model is comprised of a *database manager*, a set of *object types*, a set of *relationship types* and a set of *transactions*. The database manager performs typical database management operations including scheduling of all execution on the processor, but not necessarily including concurrency control. Database *object types* specify the structure of database objects. *Relationships* are instances of relationship types; they specify associations among the database objects and define inter-object constraints within the database. *Transactions* are executable entities that access the objects and relationships in the database. Because our testing concentrates on the object locking aspect of concurrency control, the technique described in this paper is limited to concurrency control within individual, non-related objects. The model for relationship types is described in more detail in [PDPW94].

Object Types. An *object type* is defined by $\langle N, A, M, C, CF \rangle$. The component N is the name of the object type. The component A is a set of attributes, each of which is characterized by $\langle value, time, ImpAmt \rangle$. Here,

value is a complex data type that represents some characteristic value of the object type. The *ImpAmt* field is of the same type as *value* and it represents the amount of imprecision that has been introduced into the value of a. The field $a.time$ defines the age of attribute a.

An object type's M component is a set of methods that provides the only means for transactions to access instances of the object type. A method has a set of arguments where each argument has the same structure as an attribute. An *input* argument is one whose value is used by the method to update attributes. A *return* argument is one whose value is computed by the method and returned to the invoking transaction. A method also has a known worst case execution time ($Exec$), computed using techniques described in [PE94]. The read (write) *affected set* [BR88] of a method is the set of attributes that the method reads (writes).

The C component of an object type is a set of constraints that defines correct states of an instance of the object type. An object can specify value constraints, timing constraints an imprecision constraints on its attributes.

The CF component of an object type is the *compatibility function* which uses semantic information about the methods as well as current system state ($SState$) to define compatibility between each ordered pair of methods of the object type. The function has the form:

$$CF(m_{act}, m_{req}) =< BooleanExpression >$$

where m_{act} represents a method that has an active invocation, and m_{req} represents a method for which an invocation has been requested by a transaction. In addition to specifying compatibility between two method invocations, the compatibility function also expresses information about the potential imprecision that could be introduced by interleaving the specified method invocations. The compatibility function is the means for defining semantics of locking for the semantic locking technique.

Transactions. A transaction is comprised of a set of method invocations, a set of constraints and a priority. The constraints can be expressed on execution, timing or imprecision [PDPW94]. The priority is used by the database manager to perform real-time transaction scheduling. Each method invoked by the transaction is executed at the transaction's priority.

2.2 The Semantic Locking Technique

The semantic locking technique is a concurrency control technique for database objects under the RTSORAC model. The technique uses *semantic locks* to determine which transactions may invoke methods on an object. A semantic lock gives a transaction permission to invoke a specific method on an object. The granting of semantic locks is controlled by the individual objects based on the evaluation of a set of preconditions and on the evaluation of the object's compatibility function. The technique is fully described in [DW, DW93].

In the semantic locking technique, each request for a method invocation by a transaction is associated with a request to the object for a semantic lock, which, if granted, allows the method to execute. Each object executes a mechanism called the semantic locking mechanism (SLM) when a lock is requested.

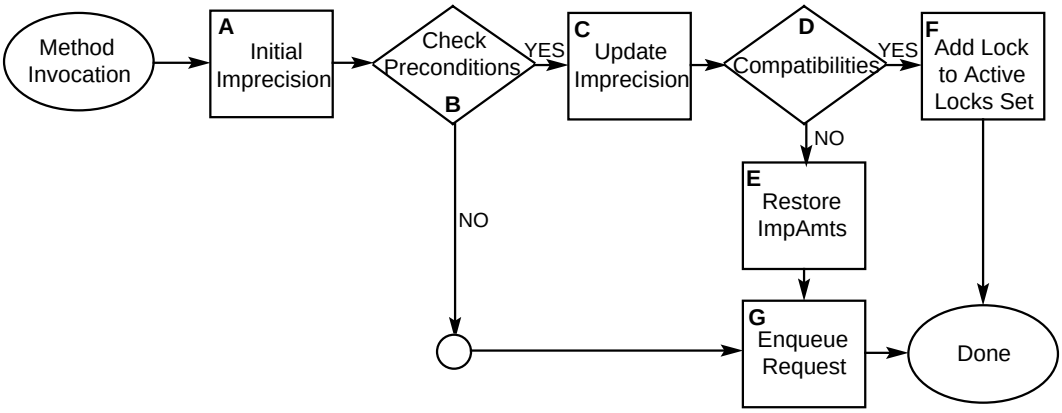

Figure 1: Semantic Locking Mechanism

There are two possible outcomes to a semantic lock request: either 1) the semantic lock becomes active and the associated method invocation is executed, or 2) the request is placed on a priority queue to be granted later. The outcome is determined by the object's SLM evaluating a set of preconditions and then evaluating the object's compatibility function. The SLM also records the amount of potential imprecision that could be introduced when concurrent semantic locks are granted. Figure 1 shows the process that the SLM performs when a semantic lock is requested by a transaction invoking a method m. We now discuss the phases of the SLM.

Initial Imprecision Calculation. The SLM first computes the potential amount of imprecision that m will introduce into the attributes that it writes and into its return arguments (Figure 1, Step A). The initial imprecision procedure computes these values by using the amount of imprecision already in the attribute or return argument and calculating how the method may update this imprecision through operations that the method performs.

Preconditions Test. The next phase of the SLM tests two preconditions that determine if granting the lock would violate temporal consistency (temporal precondition) or imprecision constraints (logical precondition) (Step B). The temporal precondition ensures that if a transaction requires temporally valid data, then an invoked method will not execute if any of the data that it reads will become temporally invalid during its execution time. The logical precondition ensures that executing the method invocation will not allow too much initial imprecision to be introduced into attributes that the method invocation writes or into its return arguments.

If any precondition fails, then the SLM places the request on the priority queue (Step G) to be retried when another lock is released. If the preconditions hold, the SLM updates the imprecision amounts for every affected attribute and every return argument (Step C). The SLM saves the original imprecision values so that they can be restored if the lock is not granted.

Compatibility Function Test. Upon m's successful passing of the preconditions, the SLM checks the compatibility function to make sure that m is compatible with all of the currently active method invocations, as well as all requested method invocations on the queue with higher priority than m (Step D). For each compatibility function test that succeeds, the SLM accumulates the imprecision that could be introduced by the corresponding interleaving. If all tests succeed, the SLM grants the semantic lock, places it in the active lock set (Step F), and allows the method to execute. If any test fails, the SLM restores the original values of any changed imprecision amounts (Step E) and places the request in the priority queue to be retried when another lock is released (Step G).

Releasing Locks. A semantic lock can be released explicitly by request of the holding transaction, or it can be released implicitly upon completion of method execution or when a transaction commits or aborts. Whenever a semantic lock is released, it is removed from the active locks set and the priority queue is checked for any requests that may be granted. Since the newly-released semantic lock may have been associated with a method invocation that restored logical or temporal consistency to an attribute, or the lock may have caused some incompatibilities, some queued requests may now be granted locks. The requests in the queue are re-issued in priority order and if any of these requests is granted, it is removed from the queue.

2.3 Implementation

We have implemented the RTSORAC model in a prototype system that extends the Open Object Oriented Database System (Open OODB) [WBT92] for real-time [WDPP94]. RTSORAC Objects are implementated in main memory using the shared memory capability. Transaction processes map the shared segment into their own virtual address spaces, thereby gaining direct access to object instances. In order to gain a lock on a method of an object in shared memory, a transaction process calls the SLM of the desired object, which determines if the method lock can be granted to the transaction.

One unique feature of our semantic locking technique is the way in which the technique defines conflict between transactions. Our user-defined compatibility function defines conflict between methods based on object semantics and system characteristics. We have implemented two versions of the semantic locking technique, each representing specific object semantics, to demonstrate how our technique can express the trade-off between temporal and logical consistency [DiP95]. The first version of the semantic locking technique, called the *semantic-logical technique*, chooses logical consistency over temporal consistency. The other version, called the *semantic-temporal technique*, chooses temporal consistency of data over logical consistency.

The semantic-logical technique is based on semantics that we have shown to be sufficient for bounding imprecision in our semantic locking technique [DW]. Certain restrictions are placed on the definition of the compatibility function to ensure that specified imprecision limits are not violated. The restrictions require the compatibility function to disallow any interleavings that would cause imprecision limits to be exceeded. The semantic-temporal technique uses the same semantics as the semantic-logical technique, except that the

Semantic-Logical	**Semantic-Temporal**
$CF(UpdateX(Y_1), UpdateX(Y_2)) =$ $(\lvert Y_1.value - Y_2.value\rvert < X.ImpLimit - X.ImpAmt)$	$CF(GetX(Y_1), UpdateX(Y_2)) =$ $(X.time < Now - 5)\ OR$ $(\lvert X.value - Y_2.value\rvert < Y_1.ImpLimit - (Y_1.ImpAmt + Y_2.ImpAmt))$

Figure 2: Compatibility Function Examples

semantic-temporal technique allows concurrency that could exceed imprecision bounds in order to preserve the temporal consistency of the data.

Note that while the semantic-temporal technique takes into account logical consistency, the semantic-logical technique ignores temporal consistency. These semantics were chosen to reflect the trade-off that must be made between temporal and logical consistency. That is, the semantic-logcial technique always requires logical consistency (within imprecision bounds). The semantic-temporal technique always chooses temporal consistency whenever a timing constraint might be violated, and falls back on logical consistency otherwise.

Figure 2 illustrates compatibility functions for the two versions of the semantic locking technique using an example involving methods that read ($GetX$) and write ($UpdateX$) an attribute X. The compatibility function for the semantic-logical technique allows two updates of the same attribute X to interleave as long as the imprecision limits of X are not violated. In the example of the semantic-temporal technique, the invocation of $UpdateX$ is allowed to interleave with the currently active invocation of $GetX$ if the return argument of $GetX$ does not become too imprecise, or if X has become temporally invalid. Notice that in this version of the semantic locking technique, logical consistency is maintained in most cases, but if temporal consistency must be restored, logical consistency may be violated.

3 Testing Environment

We utilized the prototype system described in Section 2.3 to conduct performance tests in which we compared the two versions of our semantic locking mechanism with other object-based locking techniques (exclusive locking, read/write locking and commutative locking). Each test involved generating a set of synthetic system configurations and a set of synthetic workloads. On each system configuration, we executed the corresponding workload using each of the concurrency control mechanisms. The results of these tests indicate, in general, that our semantic locking technique, in both forms, maintains transaction and data temporal consistency better than the other concurrency control techniques.

This section first defines our performance model, describing the system configuration and workload parameters, and compares it to the canonical concurrency control performance model of [ACL87]. The section goes on to describe the techniques with which we compared our semantic locking technique. Finally,

Parameter	Meaning
db_size	Number of objects
tran_size	Mean size of transactions
max_size	Size of largest transaction
min_size	Size of smallest transaction
write_prob	Probability that transaction writes object
int_think_time	Mean intratransaction think time
restart_delay	Mean transaction restart delay
num_terms	Number of terminals
mpl	Multiprogramming level
ext_think_time	Mean time between transactions
obj_io	I/O time for accessing an object
obj_cpu	CPU time for accessing an object
num_cpus	Number of CPUs
num_disks	Number of disks

Table 1: Performance Parameters for Agrawal Performance Model

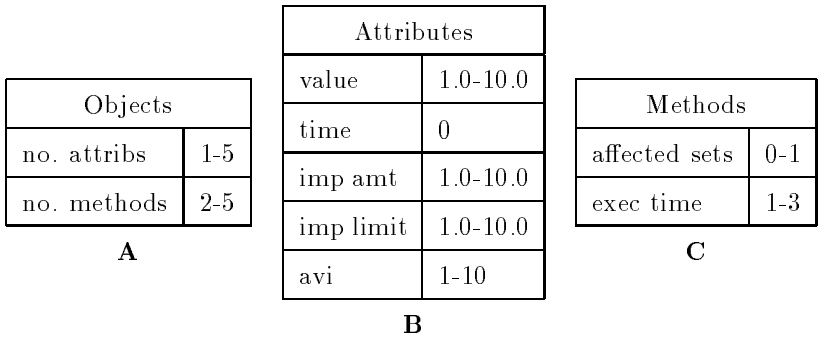

Figure 3: System Configuration Tables with Ranges for Random Values

it defines the metrics by which we measured the performance of the technqiues.

3.1 Performance Model

To test our semantic locking technique, we used the simulation model of [ACL87], which we will refer to as the Agrawal model, with several modifications to handle the required semantics and the real-time aspects of our technique. Table 1 indicates the performance parameters used in the Agrawal model. Some of the parameters in the table do not apply to our model. For instance we do not have a parameter representing the number of disks (*num_disks*), because our prototype is a main memory database. On the other hand, our model requires parameters that do not exist in the Agrawal model because our technique examines object semantics to determine concurrency control. For example, while the Agrawal model uses *db_size* to represent the size of the database, our objects have parameters representing semantics of value, time and imprecision. Also, because each object in our system configuration is different, the transactions in our workload have to specify the object and method to invoke, as well as parameters, instead of the simpler write probability (*write_prob*) of the Agrawal model.

System Configuration. The system configurations that were generated in our testing consisted of groups of data objects. Each configuration was made up of ten objects, each with randomly generated attributes, methods and constraints. The compatibility function for each object was generated based upon the concurrency control protocol used and the semantics of the object.

Figure 3 illustrates how each of the parameters in the system configuration was generated. The ranges of values for the parameters were chosen so that the system configurations were complex enough to produce interesting results, while remaining reasonable for testing purposes. Chart A shows that for each object, the number of attributes was between 1 and 5, and the number of methods was between 2 and 5. Figure 3B shows how the fields of each attribute were generated. The value field was generated randomly from the range of numbers shown. The time field of the attribute was set at the time the test started. In the chart, the zero represents the time relative to the time the object was placed in memory. The amount of imprecision initially in the attribute was chosen from a range of 1.0 to 10.0. The imprecision limit for the attribute was between 1.0 and 10.0. The *avi* (absolute validity interval) was generated from a range of relative times (1 to 10 seconds) and was then added to the absolute time that the object was placed in memory.

Figure 3C displays parameters for each method of an object. The affected sets (read affected set and write affected set) were generated randomly so that for each attribute in the object, a value of either 0 or 1 was randomly chosen to specify if the attribute was in the affected set. The execution time for each method was generated as an integer number of KiloWhetstones [DSW90]. [1]

To facilitate definition of object semantics in our testing environment, we made a simplifying assumption regarding the methods of an object:

For every attribute A in the read affected set of a method M, there is a return argument $R_{M,A}$ that returns the value read by M, and for every attribute A in the write affected set of the method M, there is an input argument $I_{M,A}$ that stores a value to be written by M. The only execution performed by a method is done by the reads and writes associated with its arguments.

This assumption provides the semantics of attribute to argument mapping thus allowing for the automatic generation of an object's compatibility function and imprecision accumulation.

The arguments of a method and their types were determined by the randomly generated affected sets. For example, if an attribute a was in the read affected set of a method m, then m had a return argument r that returned the value of a.

Workload. Generation of a workload for our performance tests involved building transactions. Each test that we performed involved 20 transactions accessing a single system configuration. Figure 4 displays the parameters that were used to build transactions for the workload. Chart A in the figure indicates that for each transaction the number of method invocations was generated randomly from a range of 1 to 5. This range of values was found to be sufficient to represent transaction of varying length. For the start time and

[1] This execution time was later converted to seconds and nanoseconds based on testing on the prototype implementation.

Transactions	
no. meth invocs	1-5
start time	4-35
deadline	12-25
exec time	computed
slack time	computed
priority	computed

A

Method Invocations	
object	sys config
method	sys config
temporal	0-1

B

Method Invocation Arguments	
imp limit	1.0-10.0
imp amt	1.0-10.0
value	1.0-10.0
time	0

C

Figure 4: Workload Tables with Ranges for Random Values

deadline of each transaction, random relative times (in seconds) were generated from the ranges indicated in the chart. They were relative to some initial starting time for the entire test. The execution time of a transaction was calculated by adding the execution times of each of the methods that the transaction invoked. The slack time was calculated by subtracting the execution time from the relative deadline. The priority of the transaction was determined based on a real-time least slack time priority assignment scheme [CSK88]. Least slack time was chosen because the priorities are static, simplifying the simulation, and because it has been shown to be optimal under certain conditions.

Figure 4B shows the parameters for each method invocation of a transaction. For the generation of each method invocation, first an object was chosen randomly from among all of the objects in the system configuration. Then a method was chosen randomly from among all of the methods of the chosen object. For each argument to the chosen method (See Figure 4C), if it was a return argument, an imprecision limit was generated randomly from a range of 1.0 to 10.0. If the argument was an input argument, a value was generated from a range of 1.0 to 10.0. These ranges were chosen to correlate with the value and imprecision ranges of the object attributes. The time field for the input argument was the time at which the write actually took place and the initial imprecision amount for the input argument was chosen from a range of 1.0 to 10.0. The generation of the method invocation also randomly determined (from 0 or 1) whether or not the transaction required temporally consistent data to be returned by the invocation.

Each transaction in a given workload requested locks using a two-phase locking scheme. The transaction requested a lock when it was needed (just before invoking the method) and the transaction held the lock until the end of its execution. Transactions that missed their deadlines were aborted and not restarted, as would be done in a firm real-time application.

3.2 Comparison Techniques

In our tests, we compared the semantic-logical technique and the semantic-temporal technique with three other object locking techniques of varying degrees of allowable concurrency - exclusive locking, read/write

Exclusive Locking	$CF(m_r, m_a) =$	$FALSE$
Read/Write Locking	$CF(m_r, m_a) =$	$(WAS(m_r) = \emptyset)\ AND\ (WAS(m_a) = \emptyset)$
Commutative Locking	$CF(m_r, m_a) =$	$(WAS(m_a) \cap (WAS(m_r) \cup RAS(m_r)) = \emptyset)$ $AND((RAS(m_a) \cap WAS(m_r)) = \emptyset)$
m_r=requested method, m_a=active method $RAS(m)$=read affected set of m, $WAS(m)$=write affected set of m		

Table 2: Compatibility Function for Comparison Techniques

locking, and commutative (affected set) locking, for a total of five concurrency control techniques. Each of the object locking techniques was implemented by defining the compatibility function accordingly. Table 2 displays the compatibility functions for each of the comparison techniques. Exclusive locking defines conflict by mutual exclusion. Only one transaction may access an object at a time. Therefore, the corresponding compatibility function is always false; allowing no methods within an object to interleave. Read/write locking of objects allows multiple readers of an object, but only one writer at a time. The compatibility function for read/write locking requires that two methods are compatible only if neither of them writes to the object. Commutativity of methods, as defined in [BR88], allows methods to interleave only if the intersections of the affected sets of the methods involved are empty. The compatibility function for commutative locking checks that if the requested method writes an attribute, no active method reads or writes the attribute, and if the requested method reads an attribute, no active method writes the attribute.

Each of the five object locking concurrency control techniques was implemented in our prototype system. For the implementation of exclusive locking, read/write locking and commutative locking we used a simplified version of the semantic locking technique in which all of the steps in the technique that involved testing or accumulation of imprecision (Steps A, B, C, and E of Figure 1) were left out because none of these techniques allows any imprecision. This removed any unnecessary overhead from the comparison techniques so that they were better represented. The temporal precondition was left in the comparison techniques so that any difference that was found among the techniques could be attributed to the way in which conflict was defined, and not to differences in how locks were acquired.

3.3 Performance Measurements

Traditionally the measure of a concurrency control protocol is the throughput of transactions [ACL87]. However, because our technique was designed for real-time applications, it is more important to measure temporal consistency than it is to measure throughput. To measure temporal consistency of transactions we examined the percentage of transactions that miss their deadlines (deadline miss ratio) [HSTR89, AGM88]. To measure the temporal consistency of the data we calculated the percentage of method requests that returned temporally invalid data to its transaction (temporal inconsistency ratio) [Son92].

Deadline Miss Ratio	Temporal Inconsistency Ratio
DL1: Vary Method Invocations	TI1: Vary Method Invocations
DL2: Vary Method Execution	TI2: Vary Method Execution
DL3: Vary Deadline	TI3: Vary Absolute Validity Interval

Table 3: Tests Performed

4 Results

For each test that we performed we generated 15 system configurations and 15 corresponding transaction sets. The results of each test were averaged over these 15 trials producing a 95% confidence level with an error of at most 8% (unless otherwise specified). We executed a test for each of the five concurrency control protocols that we compared. We also varied the interarrival time of transactions to illustrate how the techniques perform under varying system loads. We used the range of start times for a transaction as a measure of interarrival time. That is, the smaller the range of start times for a set of transactions, the closer the interarrival time and therefore the heavier the load. Table 3 summarizes the tests that we performed.

For both data temporal consistency and transaction temporal consistency we performed three test suites, each to highlight the performance of our techniques under specific conditions. The goal here was twofold. First, we wanted to show that our techniques outperform the less flexible object locking techniques over a wide range of system conditions. Second, we were interested in determining under which conditions it would be most beneficial to use semantic locking.

4.1 Deadline Miss Ratio

We performed three test suites to measure deadline miss ratio, each to highlight a particular parameter of the testing.

Test Suite DL1: Vary Method Invocations. The first test was chosen to illustrate how the length of transactions affects concurrency control. We used the number of method invocations in a transaction to represent transaction length. A short transaction had a randomly generated number of method invocations from 1 to 3. A medium length transaction had from 4 to 6 method invocations. A long transaction had from 7 to 9 method invocations.

For medium length transactions, there was a significant difference between our semantic techniques and the other techniques (Figure 5). At low contention levels, the semantic techniques performed similarly. With higher contention, the semantic-temporal technique performed better than the semantic logical technique, because with high contention, there were more chances for methods of different transactions to conflict. It was therefore more likely that the read/write conflict that allowed the semantic-temporal technique to violate logical consistency would occur. For very short transactions all of the concurrency control techniques

performed very well, missing almost no deadlines. For long transactions the semantic techniques performed slightly better than the others, with the difference diminishing as contention got higher.

The reason we see the most difference for medium length transactions is that with short transactions it is very easy for all techniques to meet their deadlines, and for long transactions, the system becomes overloaded making it difficult for any techniques to perform well.

Test Suite DL2: Vary Method Execution. The length of the methods invoked by a transaction is another way of examining the effect of length of transaction. We varied the execution time of methods so that it was randomly chosen from a range of 1 to 3 KiloWhetstones for short methods, 5 to 8 KiloWhetstones for medium length methods, and 10 to 15 KiloWhetstones for long methods.

The greatest difference between the semantic techniques and the traditional techniques is seen with short methods (Figure 6). For medium length methods, the semantic techniques missed fewer deadlines when contention was low, but as contention became higher, all of the techniques missed more deadlines, with not much significant difference among them. And for long methods, a very high percentage of deadlines were missed by all of the concurrency control techniques.

An explanation for these results is that as method execution time increased, the possibility of the real-time scheduler finding a feasible schedule (one that meets all of its deadlines) decreased. Thus, for long methods, there was very little difference among the five techniques because there were very few feasible schedules. For shorter methods, there are more feasible schedules, and the increased allowable concurrency of our semantic techniques provides the flexibility for the scheduler to find them. Thus, there was more difference among the techniques.

Test Suite DL3: Vary Deadline. We varied the length of the transactions' deadlines in order to examine how the concurrency control mechanism reacts to different real-time environments. The deadlines for transactions were randomly chosen from a range of 8 to 11 seconds for short deadlines, 12 to 15 seconds for medium deadlines, and 17 to 20 seconds for long deadlines.

The results of this test suite are illustrated by Figures 7 through 9. In each of the tests, all of the concurrency control techniques miss very few deadlines at low contention, and as contention increased, our semantic techniques performed better. When deadlines were short, the techniques all missed substantially more deadlines at very high contention, but the difference among the techniques was clear, with the semantic techniques performing better than the others. For medium and long deadlines, the semantic techniques remained virtually the same regardless of processor contention, missing almost no deadlines. The other techniques tended to miss more deadlines as processor contention increased.

These results can be explained by the fact that with medium and long deadlines, the semantic techniques provided enough concurrency for the transactions to meet most of their deadlines regardless of how many transactions were running concurrently. When processor contention was high, the traditional techniques could not provide enough concurrency for all of the executing transactions to meet their deadlines.

4.2 Temporal Inconsistency Ratio

We measured temporal inconsistency by examining the percentage of all method requests that read temporally inconsistent data. In order to do this, we had to change the system so that transactions did not abort when they missed their deadlines, but rather continued until complete. We found that if transactions were allowed to abort, a concurrency control mechanism that missed a lot of deadlines appeared to preserve temporal consistency better than a mechanism that allowed more deadlines to be made. This was because the aborted transactions stopped at a time when they were most likely to read temporally inconsistent data.

Without transaction aborts, another problem emerged. In the tests for deadline miss ratio, deadlock was avoided because transactions had a maximum amount of time to run, and then they aborted. Without aborts, there was the possibility that deadlock would occur and the tests could not be run. In order to detect a deadlock situation, we placed a deadline of two minutes on the overall test. Those transactions that were not complete after this long deadline were assumed to have been stuck in deadlock. From these transactions, we assumed that every method invocation returned temporally invalid data.

We performed three test suites (Table 3 TI1-TI3) to measure temporal inconsistency ratio. We looked at varying the number of method invocations, method execution time, and absolute validity interval. In each of these tests, the baseline parameters were as shown in Figures 3 and 4, except that the attribute initial imprecision amount always started at 0.0.

Test Suite TI1: Vary Method Invocations. With this test, we set out to determine how the length of transactions (number of method invocations) affects the amount of temporal inconsistency seen by transactions. A low number of method invocations was represented by a range of 1 to 3, medium by 4 to 6, and high by 7 to 9 method invocations.

With a small number of methods invocations, there was very little difference in temporal inconsistency among the techniques. Each technique read virtually no temporally inconsistent data because the transactions were all short enough in order to read the data before it became old.

In the tests with medium and high number of method invocations (Figures 10 and 11), the maximum error was 13%. In these cases, a significant difference between the semantic techniques and the traditional techniques emerges. Although transactions are longer, the semantic techniques allow more interleaving, which allows writing transactions to update old data and reading transactions to read the data before it expires. Another factor involved in the difference is deadlock. Recall that if a deadlock situation occurs, our tests assume that any transactions involved in the deadlock read temporally invalid data. Because the data contention in the traditional techniques was stricter than in the semantic techniques, deadlock occurred more in the traditional techniques, and therefore more temporally inconsistent data was read.

Notice the plunge in temporal inconsistency evident in Figure 11 when interarrival rate was very high. This result is unexpected because under heavy system load, we would expect more data contention and therefore higher data temporal inconsistency. However, if all transactions start at about the same time, as

they do under very high interarrival rate, they should run in approximately priority order, with very little preemption. This would take away most of the contention that might cause deadlock to occur. Thus, the decrease in temporal inconsistency at very high interarrival rate results from the lower occurence of deadlock.

Test Suite TI2: Vary Method Execution. We examined the performance of the concurrency control techniques with short method execution (1 to 3 KiloWhetstones), medium length method execution (5 to 8 KiloWhetstones), and long method execution (10 to 15 KiloWhetstones). The purpose of this test was to see how the length of the methods in the system affects the temporal consistency of the data read by transactions.

The results of all three tests (Figures 12, 13 and 14) indicate that the longer the methods, the higher the temporal inconsistency overall. Also, in general, the semantic techniques read less temporally inconsistent data than the other techniques. These results are partly due to the additional concurrency provided by the semantic techniques, and partly due to the fact that the traditional techniques are more likely to deadlock.

The drop in temporal inconsistency at very high interarrival rate is seen again here, in each of the tests. The explanation involving the lower occurence of deadlock applies here as well.

Test Suite TI3: Vary Absolute Validity Interval. We chose to test how the techniques compared under varying values for absolute temporal validity because it is this interval that defines the temporal consistency of the data. We first examined the performance when attributes were considered temporally valid for a very short period of time (low avi, 0 to 1 second). The medium absolute validity interval test chose avi from a range of 1 to 3 seconds. For the long absolute validity interval, the avi for attributes was randomly chosen from a range of 3 to 5 seconds.

When absolute validity interval is medium or long, the results were nearly identical, with the semantic techniques reading practically no temporally inconsistent data, and the traditional techniques higher. Figure 16 shows the results for long absolute validity interval. The medium interal results were very similar. With short absolute validity intervals (Figure 15), the amount of temporally inconsistent data read by each technique is greater than for medium or long absolute validity intervals and the difference among the techniques is less significant. This is because when the absolute validity interval is short, there is less time for transactions using any concurrency control technique to read valid data.

5 Conclusion

In this paper, we reviewed our model of real-time object-oriented databases and our technique for object-based real-time concurrency control and presented the results of tests comparing our technique with other, more traditional techniques. In general, the results of the tests we performed demonstrate that the increased concurrency, provided by our semantic technique, allows for better maintenance of temporal consistency constraints.

In the tests we performed to measure how transaction temporal consistency was affected, our semantic techniques usually missed fewer (never more) deadlines than the more traditional techniques. These tests further indicate that the particular implementations of the semantic locking technique that we tested are best suited in applications that have medium length transactions, short methods, or medium to long deadlines with heavy system load.

The results of the tests measuring data temporal consistency were slightly less conclusive. While it is clear that in the given testing environment, our semantic techniques perform better than the traditional techniques, it is not clear how much of that difference is due to the occurrence of deadlock. Our technique, as it currently exists, does not provide for deadlock prevention, thus, we chose to test it against other techniques with no deadlock prevention. This might seem to be an unfair comparison because techniques such as exclusive locking and read/write locking lend themselves to deadlock prevention in the form of priority ceiling protocols [SRSC91]. However, similar deadlock prevention techniques are not trivial for the other object locking concurrency control techniques that we tested.

We are currently examining the possibility of adapting the priority ceiling protocol to our semantic locking technique. At present we have produced a promising start by applying the priority ceiling protocol to affected sets [STDW96].

References

[ACL87] Rakesh Agrawal, Michael J. Carey, and Miron Livny. Concurrency control performance modeling: Alternatives and implications. *ACM Transactions on Database Systems*, 12(4):609–654, December 1987.

[AGM88] Robert Abbott and Hector Garcia-Molina. Scheduling real-time transactions: A performance evaluation. In *14th VLDB Conference*, August 1988.

[BR88] B.R. Badrinath and Krithi Ramamritham. Synchronizing transactions on objects. *IEEE Transactions on Computers*, 37(5):541–547, May 1988.

[CSK88] S. Cheng, J. Stankovic, and K.Ramamritham. Scheduling algorithms for hard real-time systems - a brief survey. In *IEEE Real-Time Systems Symposium*, pages 150–173, 1988.

[DiP95] Lisa Cingiser DiPippo. *Object-Based Semantic Real-Time Concurrency Control*. PhD thesis, Department of Computer Science, The University of Rhode Island, 1995.

[DSW90] Patrick Donohoe, Ruth Shapiro, and Nelson Weiderman. *Hartstone Benchmark User's Guide, Version 1.0*. Carnegie Mellon University, Software Engineering Institute, March 1990.

[DW] Lisa Cingiser DiPippo and Victor Fay Wolfe. Object-based semantic real-time concurrency control with bounded imprecision. *IEEE Transactions on Knowledge and Data Engineering*. To appear.

[DW93] Lisa B. Cingiser DiPippo and Victor Fay Wolfe. Object-based semantic real-time concurrency control. In *Proceedings of IEEE Real-Time Systems Symposium*, December 1993.

[HSTR89] Jiandong Huang, John Stankovic, D. Towsley, and Krithi Ramamritham. Experimental evaluation of real-time transaction processing. In *Proceedings of IEEE Real-Time Systems Symposium*, December 1989.

[PDPW94] J.J. Prichard, Lisa Cingiser DiPippo, Joan Peckham, and Victor Fay Wolfe. *RTSORAC*: A real-time object-oriented database model. In *The 5th International Conference on Database and Expert Systems Applications*, Sept. 1994.

[PE94] W. Pugh and T. Marlow (Editors). Proceedings of the ACM SIGPLAN workshop on language, compiler and tool support for real-time systems. June 1994. held in conjunction with ACM SIGPLAN PLDI Conference.

[Son92] Xiaohui Song. *Data Temporal Consistency in Hard Real-Time Systems*. PhD thesis, University of Illinois at Urbana-Champaign, 1992. Technical Report UIUCDCS-R-92-1753.

[SRSC91] Lui Sha, R. Rajkumar, Sang Son, and C. Chang. A real-time locking protocol. *IEEE Transactions on Computers*, 40(7):793–800, July 1991.

[STDW96] Michael Squadrito, Bhavani Thurasingham, Lisa Cingiser DiPippo, and Victor Fay Wolfe. Towards priority ceilings in semantic object-based concurrency control. In *1996 International Workshop on Real-Time Database Systems and Applications*, March 1996.

[WBT92] David L. Wells, José A. Blakely, and Craig W. Thompson. Architechture of an open object-oriented database management system. *IEEE Computer*, 25(10):74–82, October 1992.

[WDPP94] V. F. Wolfe, L. C. DiPippo, J.J. Prichard, and J. Peckham. The design of real-time extensions to the open object-oriented database system. In *The 1st IEEE Workshop on Object-Oriented Real-Time Dependable Systems*, Oct. 1994.

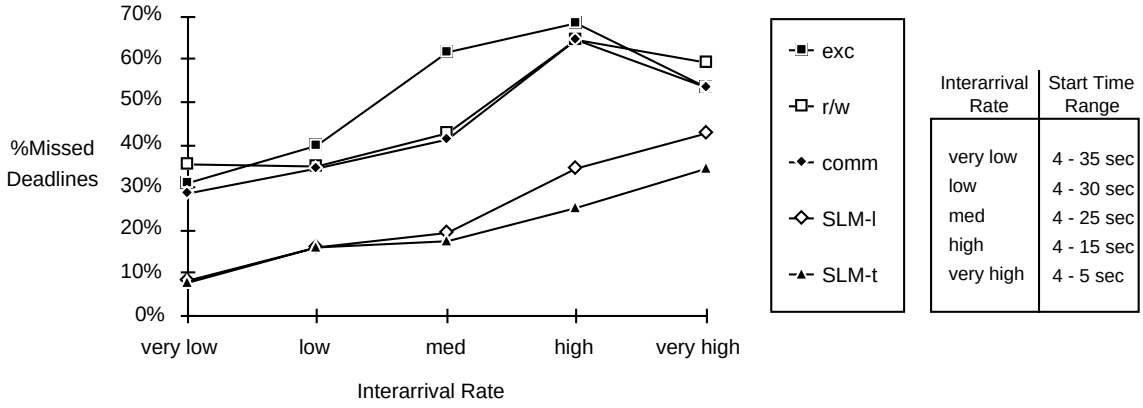

Figure 5: Missed Deadlines - Medium Method Invocations

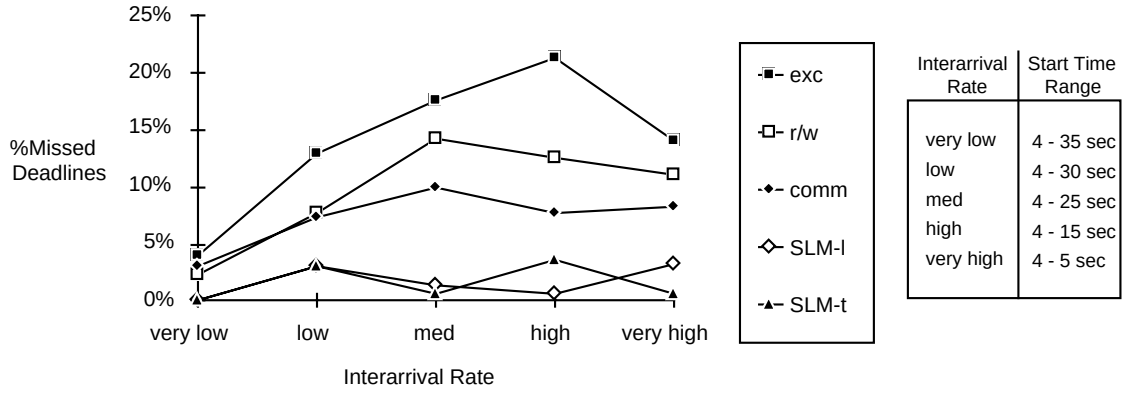

Figure 6: Missed Deadlines - Short Methods

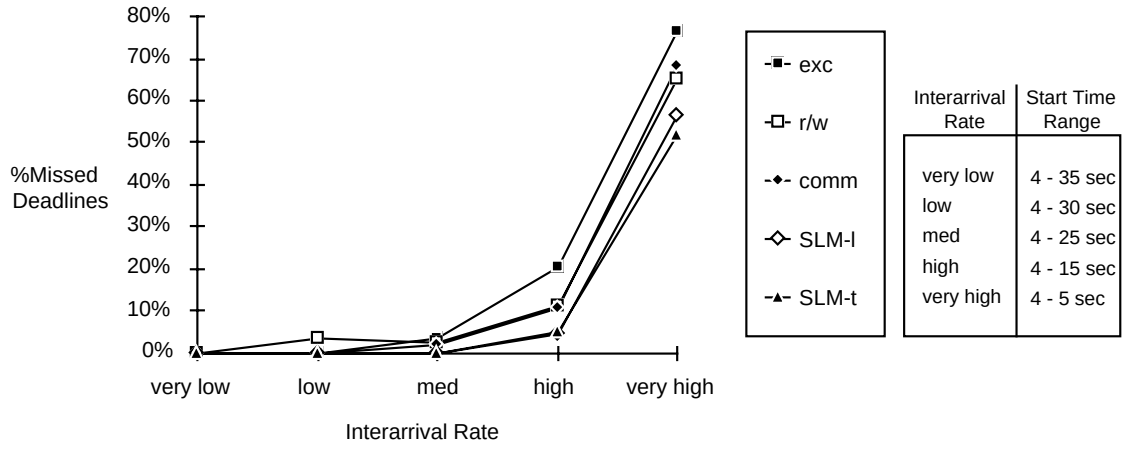

Figure 7: Missed Deadlines - Short Deadlines

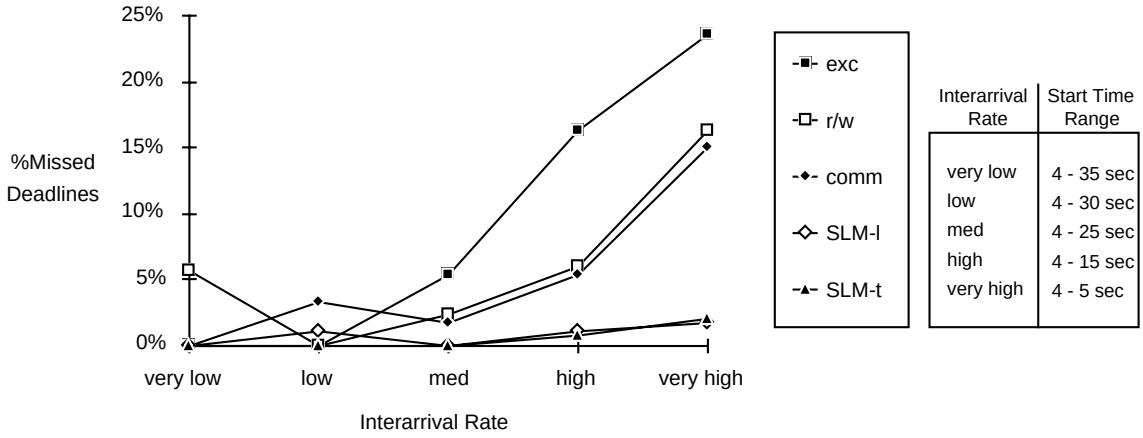

Figure 8: Missed Deadlines - Medium Deadlines

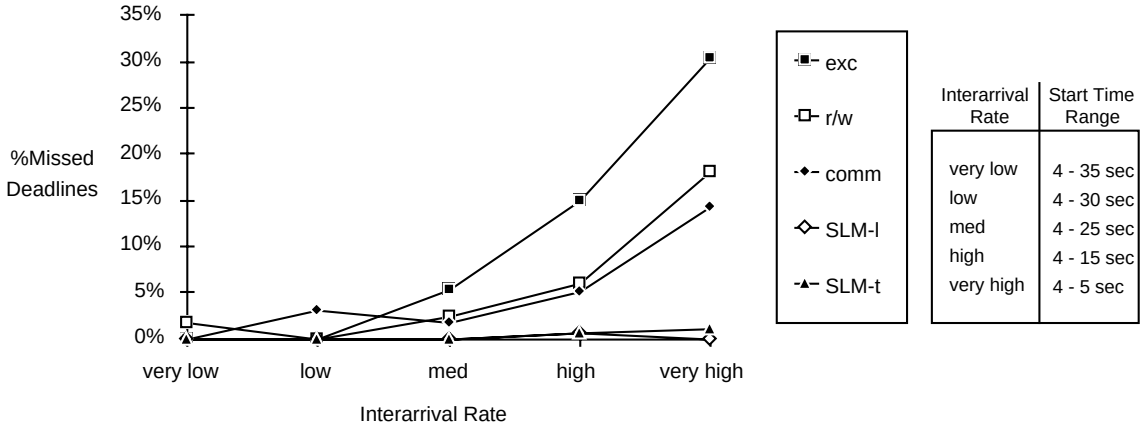

Figure 9: Missed Deadlines - Long Deadlines

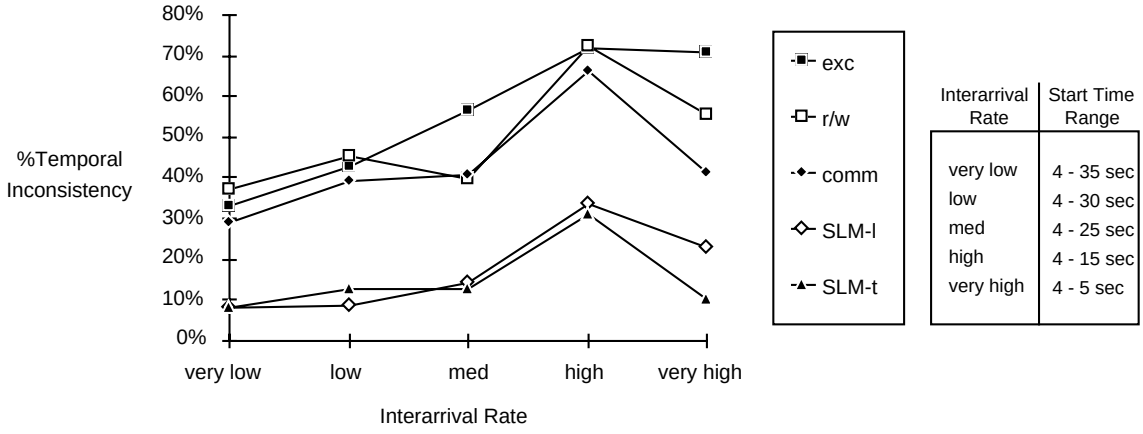

Figure 10: Temporal Inconsistency - Medium Method Invocations

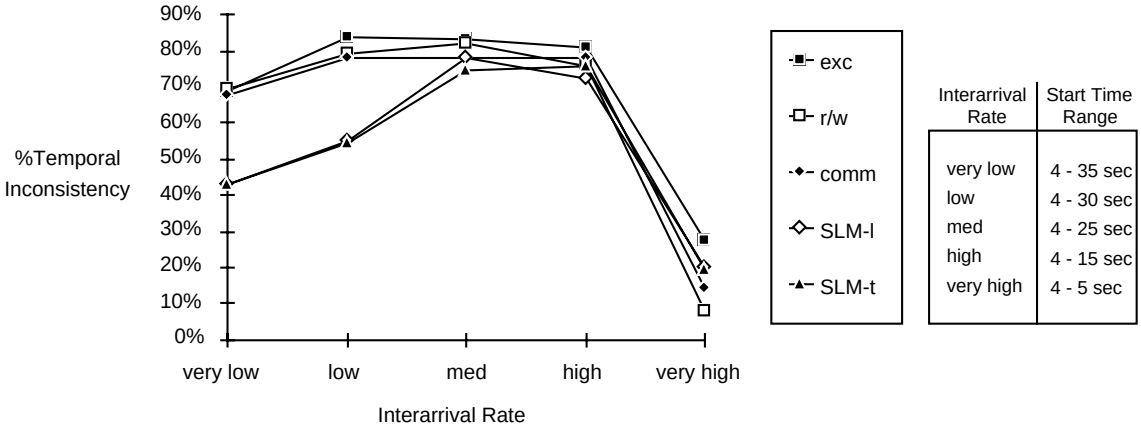

Figure 11: Temporal Inconsistency - High Method Invocations

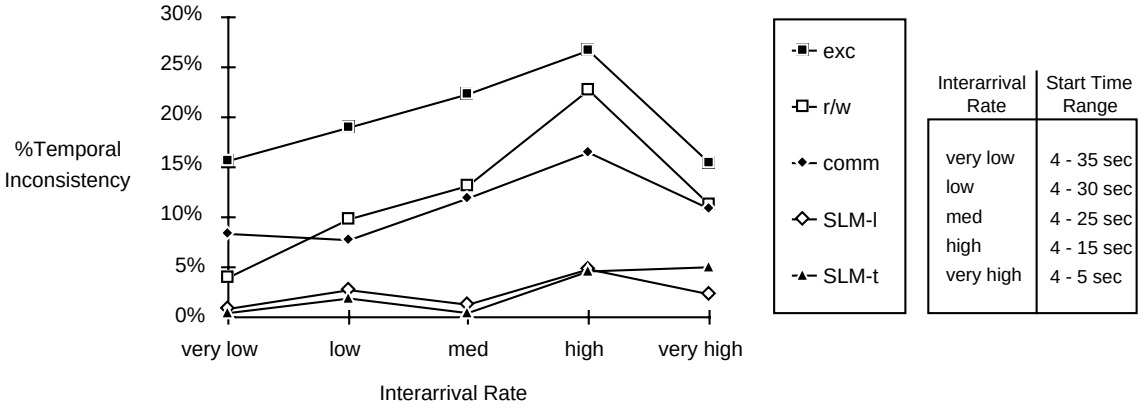

Figure 12: Temporal Inconsistency - Short Methods

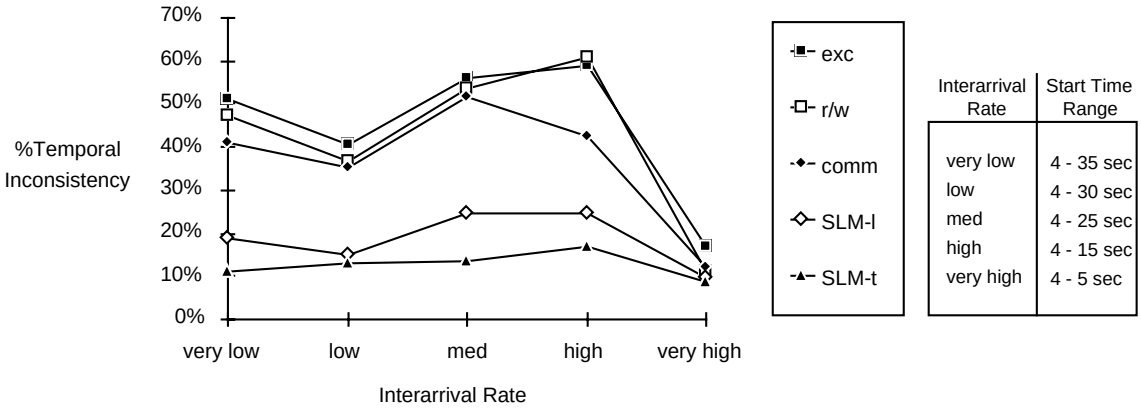

Figure 13: Temporal Inconsistency - Medium Length Methods

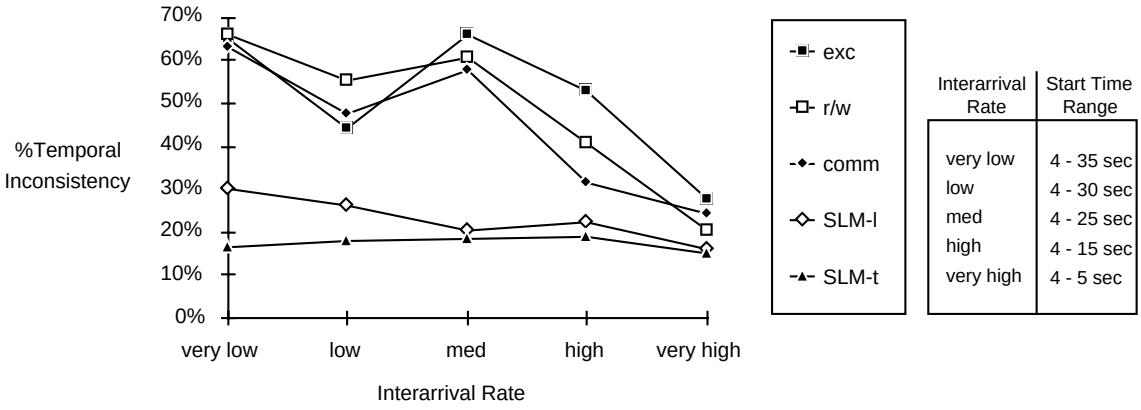

Figure 14: Temporal Inconsistency - Long Methods

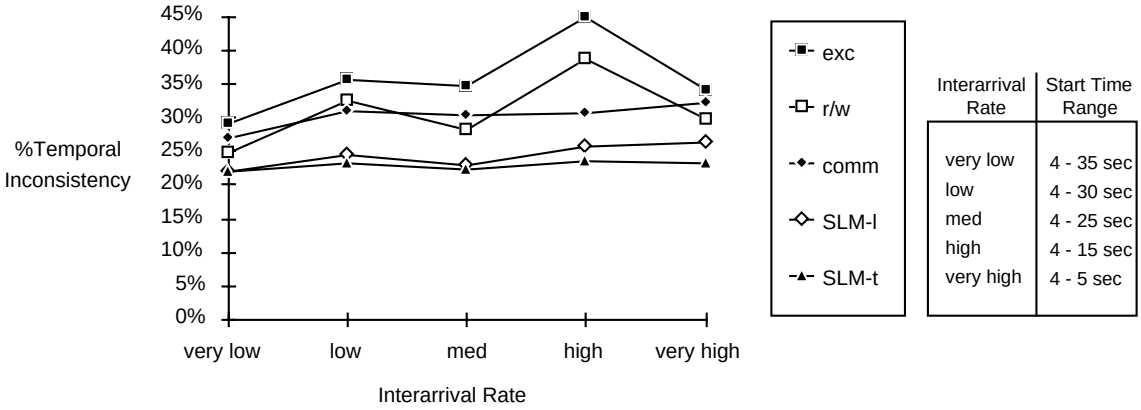

Figure 15: Temporal Inconsistency - Short Absolute Validity Interval

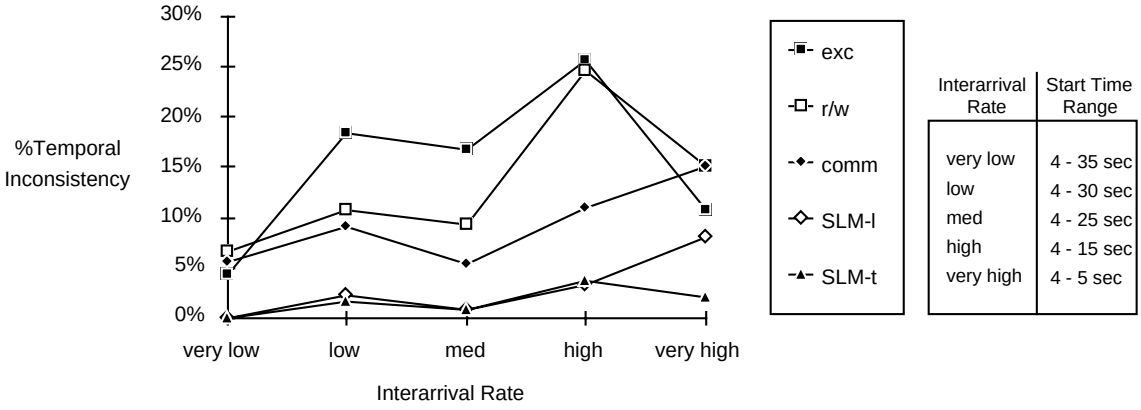

Figure 16: Temporal Inconsistency - Long Absolute Validity Interval